ELSEVIER
Science of Computer Programming 39 (2001) 31–55
Programming
www.elsevier.nl/locate/scico

# Evaluating the effectiveness of pointer alias analyses ☆

Michael Hind[a, *], Anthony Pioli[b]

[a] *IBM Watson Research Center, 30 Saw Mill River Road, Hawthorne, NY 10532, USA*
[b] *Register.com, 575 11th Avenue, New York, NY 10018, USA*

**Abstract**

This paper describes an empirical comparison of the effectiveness of six context-insensitive pointer analysis algorithms that use varying degrees of flow-sensitivity. Four of the algorithms are flow-insensitive, one is flow-sensitive, and another is flow-insensitive, but uses precomputed flow-sensitive information. The effectiveness of each analysis is quantified in terms of compile-time efficiency and precision. Efficiency is reported by measuring CPU time and memory consumption of each analysis. Precision is reported by measuring the computed solutions at the program points where a pointer is dereferenced. The results of this paper will help implementors determine which pointer analysis is appropriate for their application. 

## 1. Introduction

To effectively analyze programs written in languages that make extensive use of pointers, such as C, C++, or Java (in the form of references), knowledge of pointer behavior is required. Without such knowledge, conservative assumptions about pointer values must be made, resulting in less precise data flow information, which can adversely affect the precision and efficiency of analyses and tools that depend on this information.

A pointer alias analysis attempts to determine what a pointer can point to at compile time. As such an analysis is, in general, undecidable [27,38], approximation methods have been developed that provide trade-offs between the efficiency of the analysis and the precision of the computed solution. These analyses' worst-case time complexities

☆ This work was performed when the first author was at SUNY at New Paltz and IBM Research and the second author was at SUNY at New Paltz. This work was supported in part by the National Science Foundation under grant CCR-9633010, by IBM Research, and by SUNY at New Paltz Research and Creative Project Awards. An earlier version of this work appeared in the 5th International Static Analysis Symposium, Lecture Notes in Computer Science, 1503, pp. 57–81.

* Corresponding author.
*E-mail addresses:* hind@watson.ibm.com (M. Hind), anthony@register.com (A. Pioli).

0167-6423/01/$ - see front matter 
PII: S0167-6423(00)00014-9

range from linear to exponential. Because such worst-case complexities are often not a true indication of analysis time, algorithm designers often empirically demonstrate the efficiency of their algorithms on real programs.

Although several researchers have provided such empirical results, comparisons among results from different researchers can be difficult because of differing program representations, benchmark suites, and precision/efficiency metrics [20]. In this work, we describe an implementation of six pointer analysis algorithms (described by various researchers) that holds these factors constant, thereby focusing more on the efficacy of the algorithms and less on the manner in which the results were obtained. These analyses range in worst-case complexity from linear to polynomial.

The contributions of this paper are the following:

- a comparison of the analysis time and memory consumption of the six analyses using 24 benchmarks, ranging in size from 200 to 29,600 LOC; and
- a comparison of the direct precision of these analyses demonstrating their effectiveness using the same benchmark suite.

Our experiments hold other factors that affect precision and efficiency constant so that the results only reflect the usage of flow-sensitivity.

The rest of this article is organized as follows: Section 2 highlights the six algorithms. Section 3 describes their implementations. Section 4 describes the empirical study of the six algorithms and analyzes the results. Section 5 discussed results from other researchers. Section 6 states conclusions.

## 2. Background

Interprocedural data flow analyses can be classified according to whether they consider control flow information during the analysis [35]. A *flow-sensitive* analysis considers control flow information of a procedure during its analysis of the procedure. A *flow-insensitive* analysis does not consider control flow information during its analysis, and thus can be less precise than a flow-sensitive analysis. The goal of such an analysis is to increase efficiency.

In addition to flow-sensitivity, other factors that affect cost/precision trade-offs include whether an algorithm considers calling context, how it models the heap and aggregate objects, and which alias representation is employed. This work holds these factors constant, so that the results vary only the usage of flow-sensitivity. In particular, all analyses are context-insensitive, name heap objects based on their allocation site, collapse aggregate components, and use the compact/points-to representation (described further below).

The algorithms we consider, listed in order of increasing precision, are given below. The first five algorithms are flow-insensitive, the last of which uses precomputed flow-sensitive information. The sixth algorithm is flow-sensitive:

*AT* ("*Address Taken*"): A flow-insensitive algorithm that computes one solution set for the entire program that contains all named objects assigned to another variable.

*ST* ("*Steensgaard*"): An implementation of Steensgaard's flow-insensitive algorithm [49] that computes one solution set for the entire program and uses a union-find data structure to avoid iteration.

*AN* ("*Andersen*"): An iterative implementation of Andersen's flow-insensitive algorithm [1] that computes one solution set for entire program.

*B1* ("*Burke et al. 1*"): A flow-insensitive algorithm by Burke et al. [4,18] that computes a solution set for every function.

*B2* ("*Burke et al. 2*"): A flow-insensitive algorithm by Burke et al. [4,18] that computes a solution set for every function, but attempts to improve precision by using precomputed (flow-sensitive) kill information.

*CH* ("*Choi et al.*"): A flow-sensitive algorithm by Choi et al. [9,18] that computes a solution set for every program point.

The program being analyzed is represented as a program call (multi-) graph (PCG), in which a node corresponds to a function and a directed edge represents a potential call to the target function.[1] Each function body is represented by a control flow graph (CFG). The CH analysis uses this graph to build a simplified sparse evaluation graph (SEG) [10], which is intuitively a subset of the original CFG containing only "interesting" CFG nodes and the edges needed to connect them [20,36].

The address-taken analysis (AT) computes its solution by making a single pass over all functions in the program, adding to a global set all variables whose addresses have been assigned to another variable. These include actual parameters whose addresses are stored in the corresponding formal. Examples are statements such as "`p = &a;`", "`q = new ...;`", and "`foo(&a);`", but not simple expression statements such as "`&a;`" because the address was not stored. AT is efficient because it is linear in the size of the program and uses a single solution set, but it can be imprecise. It is provided as a base case for comparison to the other algorithms presented in this paper.

The ST analysis implements Steensgaard's algorithm [49]. One fast union/find set [51] is used to represent all alias relations, resulting in an almost linear time algorithm that makes only one pass over the program.

The AN analysis implements Andersen's context-insensitive algorithm using an iterative data flow approach rather than solving constraints as the algorithm is described [1]. The algorithm can be more precise than ST because it does not merge objects that are pointed-to by the same pointer. However, it does require iteration over all pointer-related statements. The implementation iterates only over those pointer assignment statements and function calls that do not produce constant alias relations.

The general manner in which the other three analyses compute their solutions is the same. A nested fixed point computation is used in which the outer nest corresponds to computing solutions for each function in the PCG. Each such function computation

[1] Potential calls can occur due to function pointers and virtual methods, in which the called function is not known until runtime.

```
S1:  build the initial PCG
S2:  foreach procedure, p, in the PCG, loop
S3:      initialize interprocedural alias sets of p
S4:  end loop
S5:  repeat
S6:      foreach procedure, p, in the PCG, loop
S7:          using the interprocedural alias sets (for entry of p and call sites in p),
                 compute the intraprocedural alias sets of p
S8:          using the intraprocedural alias sets of p,
                 update the interprocedural alias sets representing
                 the effect of p on each procedure that calls or is called by p
S9:      end loop
S10:     update the PCG using new function pointer aliases
S11:     foreach new procedure p added to the PCG in Step S10, loop
S12:         initialize interprocedural alias sets of p
S13:     end loop
S14: until the interprocedural alias sets and the PCG converge
```

Fig. 1. High-level description of general algorithm [18].

triggers the computation of a local solution for all program points that are distinguished in the particular analysis. For the flow-sensitive (CH) analysis, the local solution corresponds to each SEG node in the function. For the other two flow-insensitive analyses (B1, B2), the local solution corresponds to one set that conservatively represents what can hold anywhere in the function. This general framework is presented as an iterative algorithm in Fig. 1 and is further described in [18]. An extension to handle virtual methods is described in [6]. Improvements due to the use of a worklist-based implementation are reported in [20,36]. Other pointer analysis techniques are described in [5].

The B1 analysis can be more precise than the AN analysis because it can ignore some alias relations based on the scope of the variables involved in the relation, at the additional storage cost of using more than one alias set. More specifically, when computing the alias relations generated by a function, the B1 analysis removes relations involving local variables of a nonrecursive function. This filtering can improve precision at the called routine. In the AN analysis, such filtering does not occur because only one alias set is used, and thus no distinction is made between the calling and called functions.

The analyses use the *compact representation* [9,18] of representing alias relations. This representation shares the property of the *points-to* representation [13], in that it captures the "edge" information of alias relations. For example, if variable `a` points to `b`, which in turn points to `c`, the compact representation records only the following alias set: $\{\langle *a, b\rangle, \langle *b, c\rangle\}$, from which it can be inferred that $\langle ** a, c\rangle$ and $\langle ** a, *b\rangle$ are also aliases. [2]

All analyses are *context-insensitive*; they merge information flowing from different calls to the same function, and may suffer from the *unrealizable path problem* [29], i.e., they potentially propagate the aliases of the called function back to the wrong caller.

[2] See [18,31,34,36] for discussions of precision trade-offs between this representation and an explicit representation, which would contain all four alias pairs.

```
        T *p, *q, *r;
        void f1() {
S1:         p = new T;
S2:         f3(&p);
S3:         p = new T;
S4:         ... = *p;
        }

        void f2() {
S5:         q = new T;
S6:         f3(&q);
S7:         r = new T;
        }

        void f3(T** fp) {
            T a;
S8:         f4();
S9:         p = &a;
        }

        void f4() {
S10:        p = new T;
        }
```

| Analysis | aliases of *p at $S4$ |
|---|---|
| AT | a, p, q, $heap_{S1}$, $heap_{S3}$, $heap_{S5}$, $heap_{S7}$, $heap_{S10}$ |
| ST | a, $heap_{S1}$, $heap_{S3}$, $heap_{S5}$, $heap_{S10}$ |
| AN | a, $heap_{S1}$, $heap_{S3}$, $heap_{S10}$ |
| B1 | $heap_{S1}$, $heap_{S3}$, $heap_{S10}$ |
| B2 | $heap_{S1}$, $heap_{S3}$ |
| CH | $heap_{S3}$ |

Fig. 2. Example program and computed solutions. f1 and f2 are called by functions not shown.

Context-sensitive analyses [13,54] do not suffer from this problem, but may increase time/space costs. Section 4 discusses this potential imprecision.

As in [7,23–25,33,42,54], all analyses considered here represent the (possibly many) objects allocated at calls to `new` or `malloc` by creating a named object based on the CFG node number of the allocation statement. These objects are referred to as $heap_n$, where $n$ is the CFG node number of the allocation statement. These names are unique throughout the entire program. More precise heap modeling schemes [7,9,11,14,15,17,23,32,44,45] can improve precision, but may also increase time/space costs. Quantifying the effects of using context-sensitivity and various heap models is beyond the scope of this work.

Consider the simple program in Fig. 2, where functions `f1` and `f2` are called by some other functions and both call function `f3`, which calls `f4`. The AT analysis computes only one set of objects, which it assumes all pointers may point to. This set will contain eight objects, a, p, q, $heap_{S1}$, $heap_{S3}$, $heap_{S5}$, $heap_{S7}$, and $heap_{S10}$, all of which appear to be referenced at $S4$.

The ST analysis joins two objects that are pointed-to by the same pointer into one object. This leads to the joining of the points-to sets of these formerly distinct objects. This unioning removes the necessity of iteration from the algorithm. In the example, the formal parameter of `f3`, `fp`, may point to either `p` or `q`. The ST analysis therefore merges `p` and `q` as one object, resulting in a loss of distinction concerning the heap objects that either can point to. Therefore, the five objects, $a, heap_{S1}$, $heap_{S3}$, $heap_{S5}$, and $heap_{S10}$, are reported aliased to $*$`p`.

Like the ST analysis, the AN analysis computes one set of aliases that can hold anywhere in the program. However, unlike the ST analysis it does not merge objects that have a common pointer point to them, but does require iteration. This leads to `a`, $heap_{S1}$, $heap_{S3}$, and $heap_{S10}$ being reported as aliased to $*$`p`. The B1 analysis associates with every function one set, which conservatively represents what may hold at any CFG node in the function, but without considering control flow within the function. By using separate sets, it can use function scoping to eliminate spurious aliases, which leads to $heap_{S1}$, $heap_{S3}$, and $heap_{S10}$ being aliased to $*$`p` at $S4$. Because `a` is a local variable to `f3`, and thus not accessible to `f1`, the B1 analysis does not include it in the set for `f1`.

The B2 analysis attempts to improve the precision of the B1 analysis by precomputing kill information for pointers, and then uses this information during the flow-insensitive analysis at call sites. Kill information is computed in a single flow-sensitive prepass of each SEG. For each call site, $c$, two sets are computed: the set of pointers that are definitely killed on all paths from entry to $c$, and the set of pointers that are definitely killed on all paths from $c$ to exit [4,18]. For example, the precomputation will determine that all alias relations involving $*$`p` on entry to `f1` will be killed before the call to `f3` at $S2$, and thus, propagate only $\langle *p, heap_{S1}\rangle$ and $\langle *p, heap_{S3}\rangle$ to `f3`. Likewise, the analysis precomputes that all alias relations for $*$`p` returned by `f4` will be killed at the exit of `f3` because of the assignment to `p` at $S9$. Thus, the alias relation $\langle *p, heap_{S10}\rangle$ is not propagated back to `f1` or `f2`. This improves the precision over B1 by computing the set of aliases of $*$`p` at $S4$ to be $\{heap_{S1}, heap_{S3}\}$.

The CH analysis associates an alias set before ($In_n$) and after ($Out_n$) every SEG node, $n$. For example, $Out_{S1} = \{\langle *p, heap_{S1}\rangle\}$ because $*$`p` and $heap_{S1}$ refer to the same storage. The CH analysis computes $In_{S4} = \{\langle *p, heap_{S3}\rangle\}$, which is the precise solution for this simple example.

This example illustrates the theoretical precision/efficiency levels of the six analyses, from AT (least precise) to CH (most precise). The AT analysis is the most efficient analysis studied because it is linear and uses only one set. The ST analysis is almost linear. The other four analyses all require iteration, but differ in the amount of information stored: one alias set per program (AN), one set per function (B1/B2), and two sets per SEG Node (CH). [3]

[3] A specialized SEG representation can reduce the number of alias sets by 75% on average (from two per CFG node) [20,37]. This is accomplished by sharing alias sets among CFG nodes.

## 3. Implementation

The six analyses have been implemented in the NPIC system, an experimental program analysis system written in C++. The system uses multiple and virtual inheritance to provide an extensible framework for data flow analyses [21,36]. A prototype version of the IBM VisualAge C++ compiler [26] is used as the front end. The abstract syntax tree constructed by the front end is transformed into a PCG and a CFG for each function, which serve as input to the analyses. No CFG is built for library functions. We model (by hand) a call to a library function based on its semantics, thereby providing the benefits of context-sensitive analysis of such calls. Library calls that cannot affect the value of a pointer are treated as the identity transfer function. Array elements and field components are not distinguished. The implementation also assumes that pointer values will only exist in pointer variables, and that pointer arithmetic does not result in the pointer going beyond array boundaries. As stated in Section 2, heap objects are named based on their allocation site. The implementation handles `setjmp/longjmp` in a manner similar to Wilson [53]; all calls to `setjmp` are recorded and used to determine the effects of a call to `longjmp`.

An alias set class that implements the compact representation is used to represent alias relations for the AN, B1, B2, and CH analyses. The B1, B2, and CH analyses are implemented using worklists. An earlier iterative implementation of CH is discussed in [20,36]. All implementations incorporate function pointer analysis into the pointer alias analysis by building the PCG in an optimistic manner.

To model the values passed as `argc` and `argv` to the `main` function, a dummy `main` function is added, which then calls the benchmark's `main` function, thus simulating the effects of `argc` and `argv`. This function also initializes the `_iob` array, used for standard I/O. The added function is similar to the one added by Ruf [39,41] and Landi et al. [28,30]. Initializations of global and static variables are automatically modeled as assignment statements in the dummy `main` function.

## 4. Results

This section provides empirical evidence of the efficiency and precision for the six algorithms discussed in Section 2. The results were collected on a 333 MHz IBM RS/6000 PowerPC 604e with 512MB RAM and 1GB paging space, running AIX 4.1.5. The executable was built with IBM's xlC compiler using the "-O3" option.

Our benchmark suite contains 24 C programs, 21 provided by other researchers [13,30,39,43] and 3 from the SPEC CINT92 [3] and CINT95 [48] benchmarks.[4] Table 1 describes characteristics of the suite. The third column contains the number of lines in

[4] Because we are using a C++ front end, some programs had to be syntactically modified to satisfy C++'s stricter typechecking semantics. A few program names are different than those reported by Ruf [39]. Ruf referred to ks as part [41] and the SPEC CINT92 program 052.alvinn was named backprop in one suite Ruf used [2].

Table 1
Benchmark suite and static characteristics

| Name | Source | LOC | CFG nodes | Fcts | Call sites | | Ptr-Asgn nodes (%) | Rec fcts | Alloc sites |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | User | Lib | | | |
| allroots | Landi | 227 | 159 | 7 | 19 | 35 | 1.3 | 2 | 1 |
| 052.alvinn | SPEC92 | 272 | 229 | 9 | 8 | 13 | 10.0 | 0 | 0 |
| 01.qbsort | McCat | 325 | 170 | 8 | 9 | 25 | 24.1 | 1 | 5 |
| 06.matx | McCat | 350 | 245 | 7 | 18 | 37 | 13.5 | 0 | 9 |
| 15.trie | McCat | 358 | 167 | 13 | 19 | 21 | 23.4 | 3 | 5 |
| 04.bisect | McCat | 463 | 175 | 9 | 11 | 18 | 9.7 | 0 | 2 |
| fixoutput | PROLANGS | 477 | 299 | 6 | 12 | 85 | 4.4 | 0 | 3 |
| 17.bintr | McCat | 496 | 193 | 17 | 27 | 28 | 8.8 | 5 | 1 |
| anagram | Austin | 650 | 346 | 16 | 22 | 38 | 9.5 | 1 | 2 |
| lex315 | Landi | 733 | 569 | 17 | 102 | 52 | 6.5 | 0 | 3 |
| ks | Austin | 782 | 526 | 14 | 17 | 67 | 27.4 | 0 | 5 |
| 05.eks | McCat | 1202 | 677 | 30 | 62 | 49 | 4.0 | 0 | 3 |
| 08.main | McCat | 1206 | 793 | 41 | 68 | 53 | 20.9 | 3 | 8 |
| 09.vor | McCat | 1406 | 857 | 52 | 174 | 28 | 28.6 | 5 | 8 |
| loader | Landi | 1539 | 691 | 30 | 79 | 102 | 8.8 | 2 | 7 |
| 129.compress | SPEC95 | 1934 | 17 012 | 25 | 35 | 28 | 0.2 | 0 | 0 |
| ft | Austin | 2156 | 775 | 38 | 63 | 55 | 18.6 | 0 | 5 |
| football | Landi | 2354 | 2854 | 58 | 257 | 274 | 1.8 | 1 | 0 |
| compiler | Landi | 2360 | 1767 | 40 | 349 | 107 | 5.1 | 14 | 0 |
| assembler | Landi | 3446 | 1845 | 52 | 247 | 243 | 16.6 | 0 | 16 |
| yacr2 | Austin | 3979 | 2070 | 59 | 158 | 169 | 6.6 | 5 | 26 |
| simulator | Landi | 4639 | 2929 | 111 | 447 | 226 | 6.3 | 0 | 4 |
| flex | PROLANGS | 7659 | 7107 | 88 | 375 | 239 | 5.2 | 4 | 10 |
| 099.go | SPEC95 | 29 637 | 31 788 | 373 | 2054 | 22 | 1.5 | 1 | 0 |
| Average | | | | | | | 11.0 | | |

the source and header files reported by the Unix utility `wc`. The fourth column reports the number of CFG nodes, which include nodes created by the initialization of globals. Assignment statements are created for both implicit and explicit initializations. The large number of CFG nodes for the `129.compress` benchmark is due to many such array initializations. The fifth column reports the number of user-defined functions (nodes in the PCG), which includes the dummy `main` function. The two columns marked "Call Sites" give the number of call sites, distinguished between user and library function calls. The last column reports the percentage of CFG nodes that are considered as pointer-assignment nodes. The analysis treats an assignment as a pointer-assignment if the variable involved in the pointer expression on the left side of the assignment is declared to be a pointer.[5] The last two columns report the number of recursive functions (functions that are in PCG cycles) and heap-allocation sites in each program. The last row of the table reports the average pointer-assignment

[5] This is more conservative than considering statements in which the left side *expression* is a pointer. Thus, statements such as "p->field = ..." are treated as pointer assignments no matter how the type of field is declared. A more accurate categorization would not affect the precision of the analysis, but could improve the efficiency by reducing the number of nodes considered during the analysis [20,36].

node percentage, which is computed by averaging the corresponding value over the 24 benchmarks.

### 4.1. Efficiency results

To measure efficiency we report the analysis time and the maximum memory usage for each benchmark and analysis. The analysis time, reported in seconds, is the time spent in each alias analysis, which includes any analysis-specific preprocessing, such as building the SEG from the CFG in the CH analysis. The times do not include the time to build the initial PCG and CFGs because this time is constant for all analyses. This information is displayed in the bar chart in Fig. 3.

Both the AT and ST analyses are efficient; they required less than 1 second for all programs. The AN analysis completed in less than 10 seconds on all programs. The AN analysis was incomparable to the B1 analysis, in some cases it was faster (almost three times faster on `flex`) and in others it was slower (over twice as slow on `099.go`). The AN analysis can be more efficient than B1 because it uses only one alias set. However, because the B1 analysis tracks alias sets specific to each function, it can limit alias relations based on function scoping, which can improve the performance of the alias queries. Further investigation is required to verify the causes for the variance in analysis time of AN and B1. The B2 analysis is always slower than the B1 analysis (as expected) and is sometimes slower than the CH analysis (`099.go`). The CH analysis, which in some cases is comparable to the AN and B1 analyses, can also be almost three times slower than these analyses (`flex`).

Another conclusion from Fig. 3 is that analysis time is not only a function of program size; it also depends on the amount of alias relation propagation along the PCG and SEGs. Table 2 illustrates this point for the CH analysis. For example, `099.go`, despite being our largest program, is analyzed at one of the fastest rates, while `flex`, the second largest program, is analyzed at one of the slowest rates.

A more precise and time-consuming alias analysis may not be as inefficient as it may appear because the time required to obtain increased precision may reduce the time required by subsequent analyses that utilize mod-use information, and thus pointer alias information, as their input [21,46]. This can also be true about pointer alias analysis itself, which also utilizes pointer alias information during its analysis.

Fig. 4 reports the maximum memory usage during the analysis process minus the initial storage (the memory required for the intermediate representation, statistics-related data, and empty alias relation data structures). Listed next to each benchmark name is the initial storage in MBs. This storage is roughly proportional to the size of the program. The information was obtained by using the "`ps v`" command under `AIX 4.1.5`.

The results from Fig. 4 show that the memory consumption of the CH analysis can be several times larger than the other analyses for large programs (`099.go`) or programs that make heavy use of pointers (`08.main`, `09.vor`, and `flex`). One would expect the memory usage for the CH analysis to be an order of magnitude larger than the

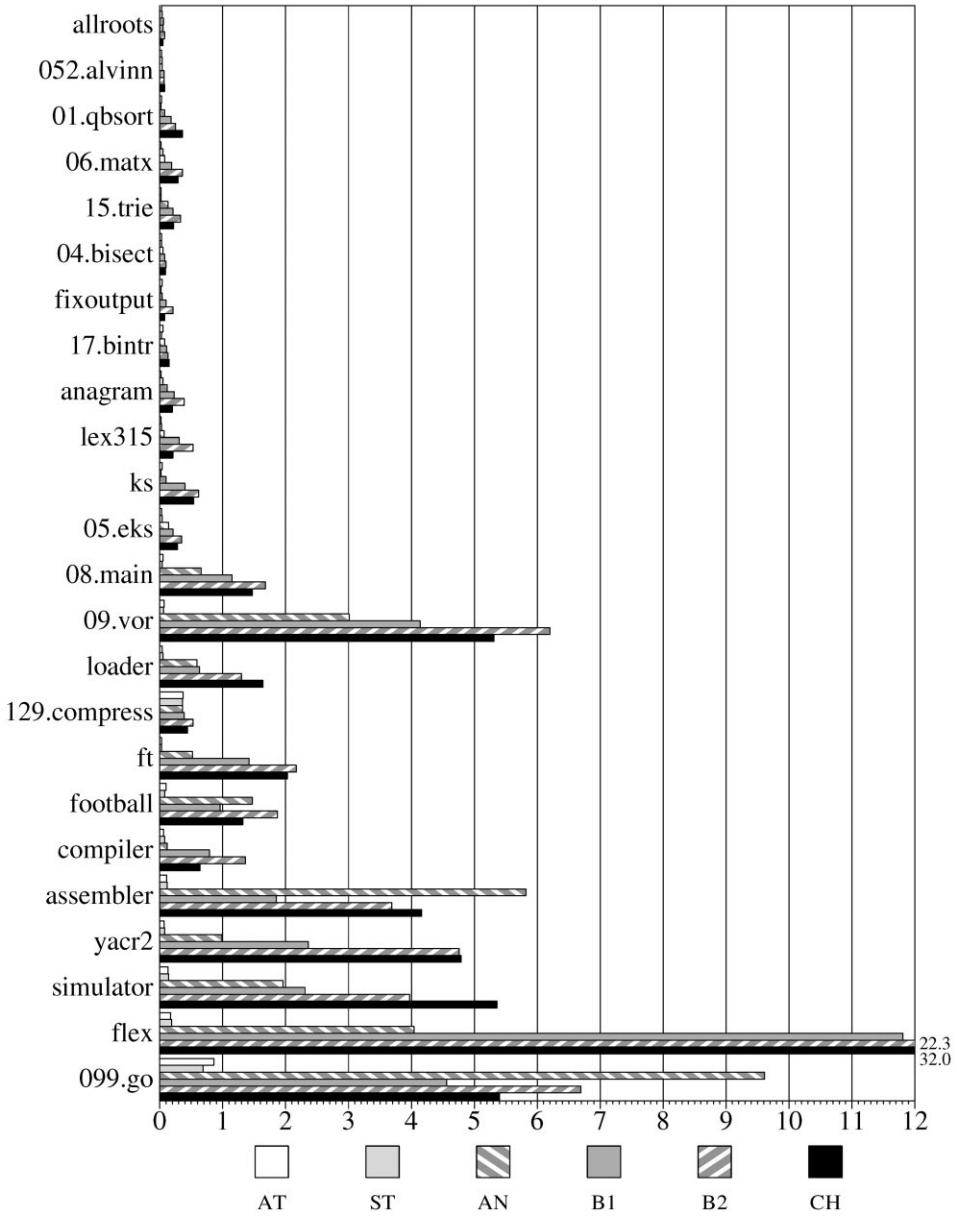


Fig. 3. Analysis time in seconds.

B1 analysis because the CH analysis can use many more alias sets. (The CH analysis *potentially* uses two alias sets for every CFG node in the program plus two alias sets for each function. The B1 analysis uses two alias sets for each function.) Although the initial implementation did have this property, several storage saving schemes, such as using a SEG, significantly reduced the storage requirements and analysis time of the

Table 2
Flow-sensitive analysis speed, computed by dividing the size of the program (as measured in LOC or CFG Nodes) by the analysis time

| Program | LOC | CFG nodes | CH time | LOC/s | Nodes/s |
|---|---|---|---|---|---|
| allroots | 227 | 159 | 0.05 | 4540 | 3180 |
| 052.alvinn | 272 | 229 | 0.08 | 3400 | 2863 |
| 01.qbsort | 325 | 170 | 0.36 | 903 | 472 |
| 06.matx | 350 | 245 | 0.29 | 1207 | 845 |
| 15.trie | 358 | 167 | 0.22 | 1627 | 759 |
| 04.bisect | 463 | 175 | 0.09 | 5144 | 1944 |
| fixoutput | 477 | 299 | 0.08 | 5963 | 3738 |
| 17.bintr | 496 | 193 | 0.15 | 3307 | 1287 |
| anagram | 650 | 346 | 0.20 | 3250 | 1730 |
| lex315 | 733 | 569 | 0.21 | 3490 | 2710 |
| ks | 782 | 526 | 0.54 | 1448 | 974 |
| 05.eks | 1202 | 677 | 0.28 | 4293 | 2418 |
| 08.main | 1206 | 793 | 1.47 | 820 | 539 |
| 09.vor | 1406 | 857 | 5.31 | 265 | 161 |
| loader | 1539 | 691 | 1.64 | 938 | 421 |
| 129.compress | 1934 | 17 012 | 0.44 | 4395 | 38 664 |
| ft | 2156 | 775 | 2.03 | 1062 | 382 |
| football | 2354 | 2854 | 1.32 | 1783 | 2162 |
| compiler | 2360 | 1767 | 0.64 | 3688 | 2761 |
| assembler | 3446 | 1845 | 4.16 | 828 | 444 |
| yacr2 | 3979 | 2070 | 4.79 | 831 | 432 |
| simulator | 4639 | 2929 | 5.36 | 865 | 546 |
| flex | 7659 | 7107 | 32.00 | 239 | 222 |
| 099.go | 29 637 | 31 788 | 5.40 | 5488 | 5887 |
| Average | | | | 2491 | 3148 |

CH analysis without affecting precision [20,36]. It is not clear if these techniques will keep the storage requirements of CH comparable with B1 when larger programs are analyzed.

Although it is more difficult to characterize the memory consumption of the other analyses, on programs with a significant amount of alias relations (`08.main`, 09.vor, and `flex`), there does appear to be a difference between those analysis with one alias set (AT, ST, AN) and those with one per function (B1, B2). For these programs, having only one alias set is an advantage in memory usage unless the precision of the analysis results in a significantly larger number of relations in this set as compared to the specialized set for each function. However, for these extra relations to outweigh program size, the difference in the size of the sets would need to increase proportionally with the number of functions.

### *4.2. Precision results*

To collect precision information, the system traverses the representation visiting each expression containing a pointer dereference and, using the computed alias information,

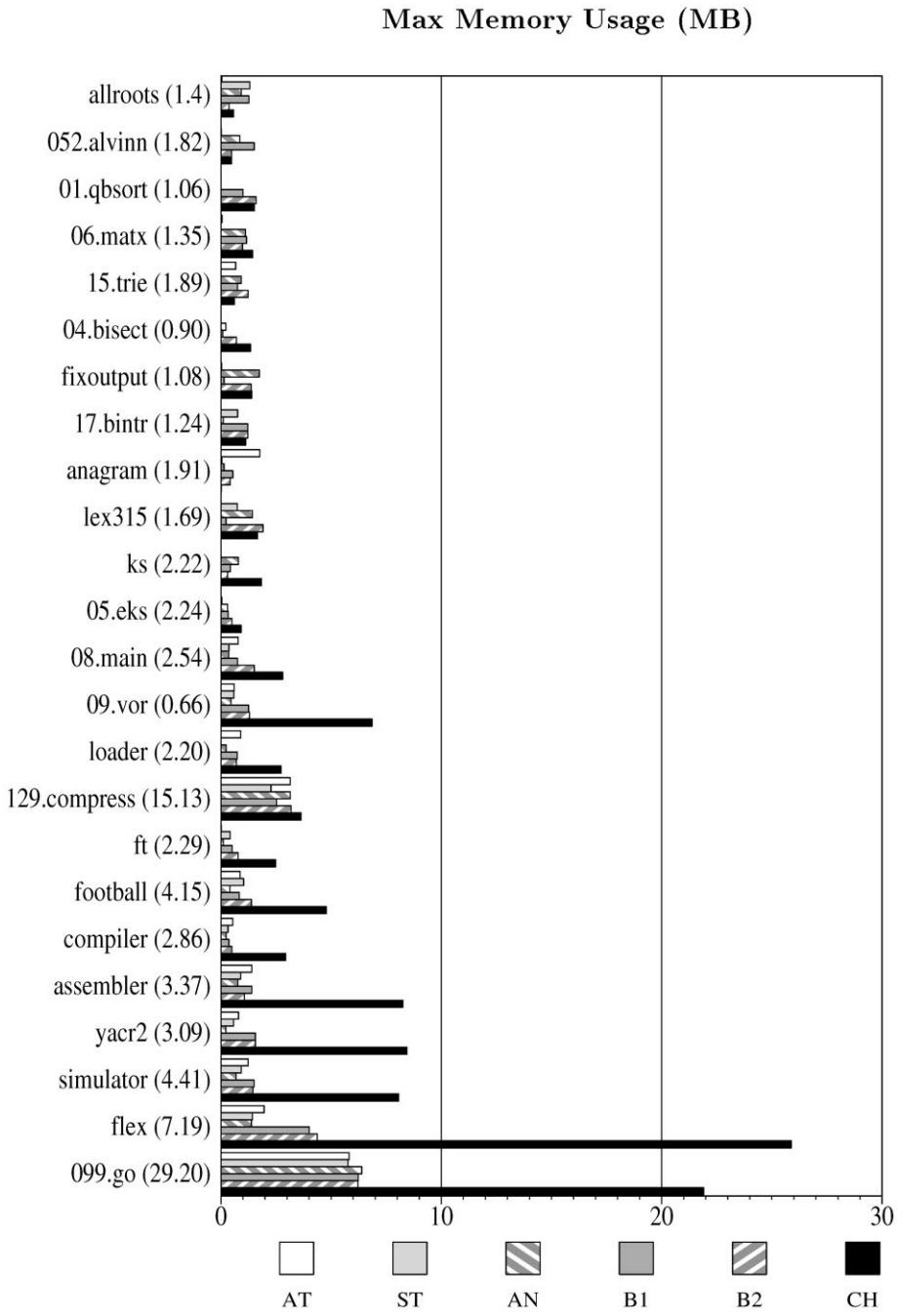


Fig. 4. Maximum memory usage for pointer analysis only. The number next to each benchmark is the initial memory required for the intermediate representation, statistics-related data, and empty alias relation data structures. This number is not included in the bar chart.

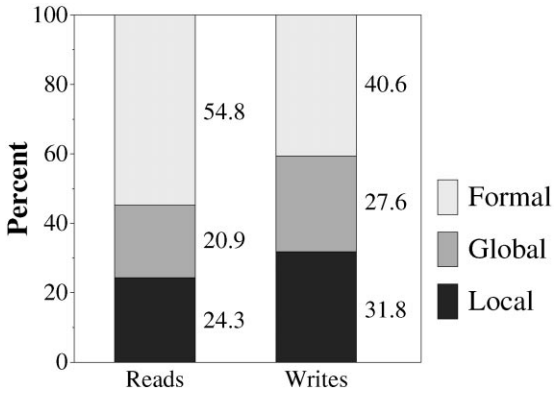


Fig. 5. Classification of dereferenced pointer types for all programs.

reports how many named objects are aliased to the pointer expression. We report the average number of such dereferences for both reads and writes. This form of counting provides a precision metric based on the use of alias information, and therefore can be more meaningful than recording the average alias set size, i.e., the average number of aliases at all program points.

A pointer expression with multiple dereferences, such as `***p`, is counted as multiple dereference expressions, one for each dereference. The intermediate dereferences (`*p` and `**p`) are counted as reads. The last dereference (`***p`) is counted as a read or write depending on the context of the expression. Statements such as `(*p)++` and `*p += increment` are treated as both a read and a write of `*p`.

We consider a pointer to be dereferenced if the variable is declared as a pointer or an array formal parameter, and one or more of the "*", "->", or "[ ]" operators are used with that variable. Formal parameter arrays are included because their corresponding actual parameters could be a pointer. We do not count the use of the "[ ]" operator on arrays that are not formal parameters because the resulting "pointer" (the array name) is constant, and therefore, counting it may skew results. Fig. 5 classifies the type of pointer dereferenced averaged over all programs.

The manner in which runtime objects are summarized must be considered in evaluating precision results. For example, a model that uses several names for objects in the heap may seem less precise when compared to a model that uses fewer names even though the percentage of the heap accessed may be greater [39]. Similarly, analyses that represent invisible objects (objects not lexically visible in the current procedure, such as locals of a calling routine) or string literals as single objects may report fewer objects.

Our analyses distinguish heap objects based on their allocation site, represent each invisible object distinctly using its name, and model all string literals using one object. The modeling of string literals improves efficiency at the cost of precision, and differs from [5,20], where each string literal is modeled as a separate object.

Assuming a correct input program, each pointer dereference should correspond to at least 1 object at runtime, and thus 1 serves as a lower bound for the average number of objects aliased to a pointer expression. Although a precision result close to 1 demonstrates the analysis is precise (modulo heap and invisible object naming), a larger number could reflect an imprecise algorithm, a limitation of static analysis, or a pointer dereference that corresponds to different memory locations over the program's execution.

The two charts of Fig. 6 provides a graphical layout of precision information for reads and writes through a dereferenced pointer. Next to each benchmark is the total number of such reads or writes. For each benchmark only four bars are presented; the results for AN, B1, and B2 are combined into one bar because the precision results are exactly the same for these analyses. The last group of bars presents the average over all benchmarks' averages. The AT analysis is always considerably less precise than the other analyses. On average the AT set contains 29.67 objects for reads and 30.52 objects for writes.[6] As one would expect this set to increase with the size of the program, the precision for this analysis will worsen with larger programs. These results suggest that the precision of this simple analysis may not be acceptable.

The ST analysis matches the precision of the B1 analysis in 7 of 24 programs for reads and 4 of 24 programs for writes. However, when considering only programs larger than 1000 LOC these values are 2 of 19 (reads) and 0 of 19 (writes). Thus, for programs of significant size it appears there will be a difference in precision between the ST analysis and a more precise flow-insensitive analysis. However, given the significant efficiency advantages of the ST analysis over the more precise analyses, a loss of precision may be acceptable, particularly for programs that are too large to run the more precise analyses. Furthermore, the significance of the precision difference depends on the how the alias information is used.

The AN and B1 analyses are identical in precision on all programs. Thus, the theoretical increase in precision B1 offers over AN, killing of alias relations involving variables local to a nonrecursive function at the end of the function, was not seen in practice. The precision of B2 is also the same as B1. One explanation may be that an alias relation created to simulate a reference parameter, in which the formal points to the actual, typically is not killed in the called routine, i.e., the formal parameter is not modified, but rather is used to access the passed actual. Thus, programs containing these alias relations will not benefit from the precomputed kill information.

The AN/B1/B2 analyses match the precision of the CH analysis in both reads and writes in 18 of 24 programs.[7] Two other programs differ only in average read or write, but not both. Averaging the benchmarks' averages shows only a marginal improvement in precision for the CH analysis over the AN/B1/B2 analyses, 2.29 to 2.34 for reads, and 2.01 to 2.05 for writes. This seems to suggest that the added precision obtained

[6] The numbers differ because the compiler benchmark did not have any writes through a pointer, but did have five reads.

[7] Prior results [5,20] reported more programs with differences because strings were modeled as individual objects.

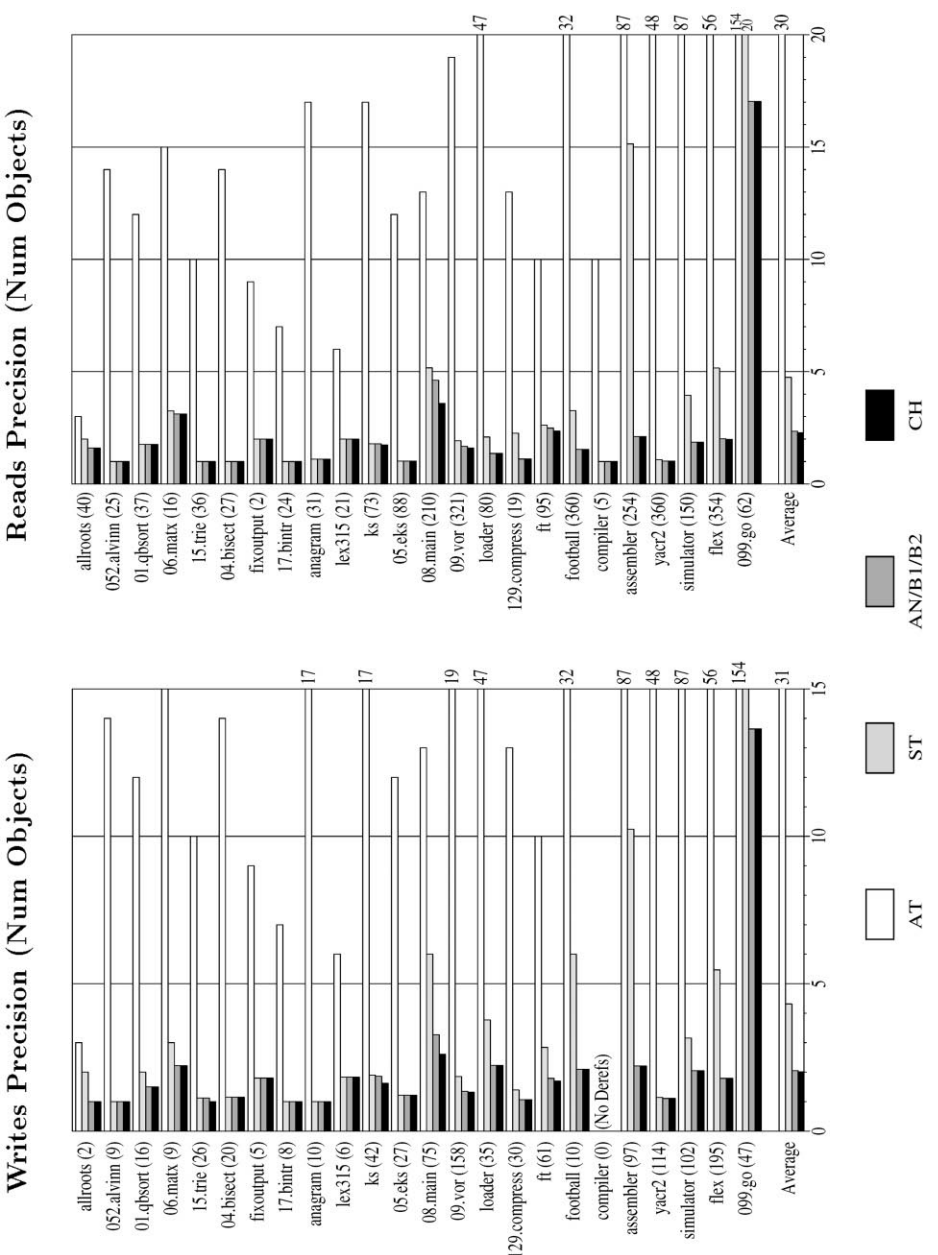


Fig. 6. Precision results.

by the CH analysis in considering control flow within a function is not significant for those benchmarks, at least where pointers are dereferenced. We offer two possible explanations:

(i) Pointer variables are often not assigned more than one distinguished object within the same function. Thus, distinguishing program points within a function, a key difference between the CH and AN/B1/B2 analyses, does not often result in an increase in precision. We have seen exceptions to this in the function `InitLists` of the `ks` benchmark and in the function `InsertPoint` in the `08.main` benchmark. `InitLists` uses the same list pointer in two list-creating loops.

In `InsertPoint` the same `tmp` pointer is used to traverse a list and to create a new list. In the list creation code the pointer is dereferenced to initialize the created node. The CH analysis reports that this can only be the heap object that was just created, while the AN/B1/B2 analyses report the possible nodes that the `tmp` pointer pointed to during the earlier list traversal. This accounts for most of the precision differences with write dereferences in the `08.main` benchmark.

The main cause of the differences in read dereferences in `08.main` is some peculiar code in the `main` function, in which the function `Draw_All` is called with the variable `o`, a pointer to an `object`, being passed as a parameter. In the previous statement, this variable is assigned the value of another variable which is always NULL. Thus, NULL is passed to this call. As this is the only call to this function, the CH analysis can determine that `Draw_All` and all functions it calls passing its parameter, will have NULL as the value of the pointer. However, the variable `o` is also used earlier in `main` to point to some object, which points to other objects. Thus, at the call to `Draw_All` the AN/B1/B2 analyses do not have the benefit of the killing definition to `o` and report that it can point to other objects besides NULL.[8] Without this function call in `main` the precision for the B1 analysis is much closer to the CH analysis.

(ii) It seems that a large number of alias relations are created at call sites because of actual/formal parameter bindings. The lack of a substantial precision difference between the CH and AN/B1/B2 analyses may be because these algorithms rely on the same (context-insensitive) mapping mechanism at call sites.

Figs. 7 and 8 further refine the precision information by decomposing each bar in Fig. 6 into the average object type pointed to. For example, of the 1.86 objects that `simulator` points to on average for read dereferences, 0.37 of these objects are nonvisible locals, 0.38 are globals, 0.78 are formal parameters, and 0.33 are (synthetic) heap locations.

[8] The B2 analysis suffers the same imprecision because the killing information it computes removes only aliases involving o coming from the entry set of main (none exist), but then adds *all* aliases generated in main.

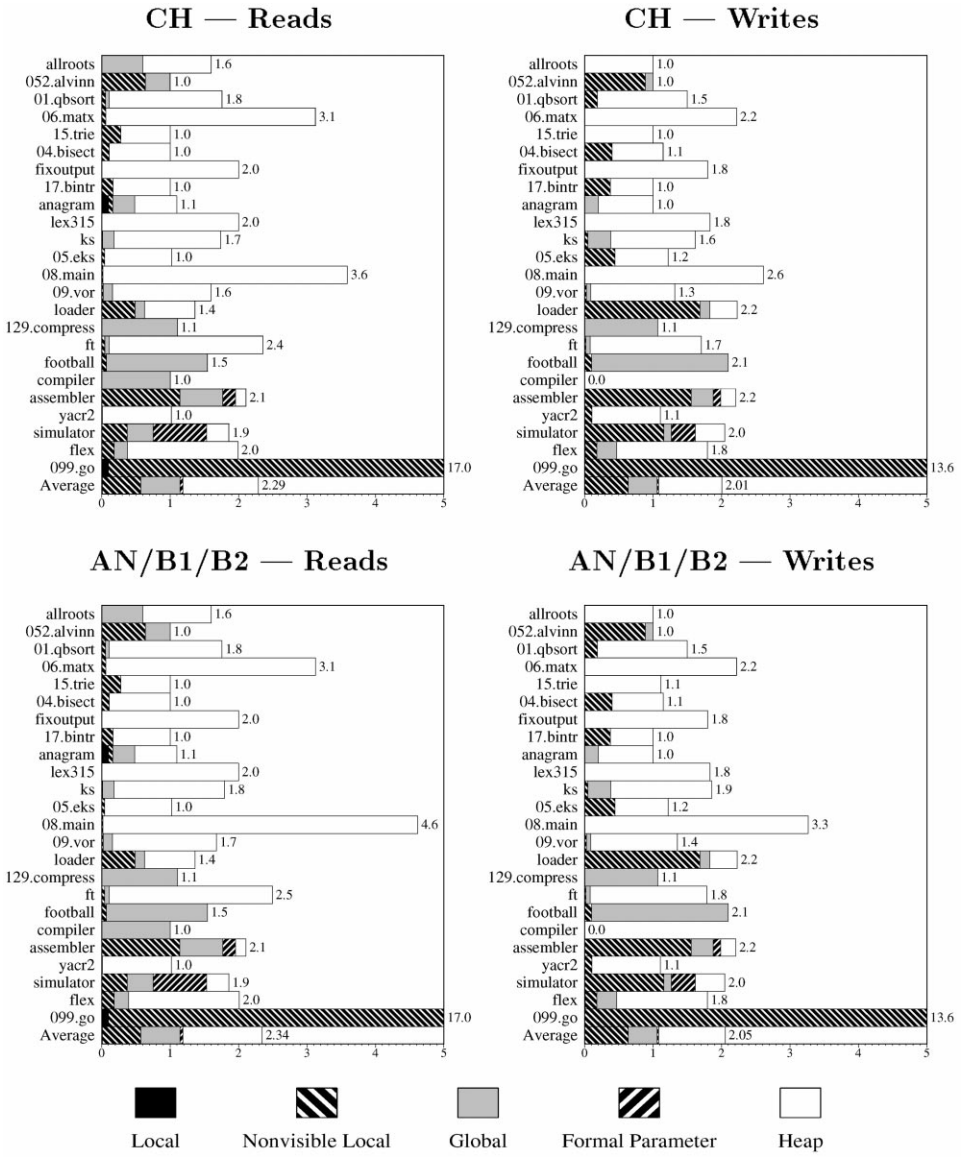


Fig. 7. Breakdown of average object type pointed to by a dereferenced pointer for the CH and AN/B1/B2 analyses. The bars for 099.go are truncated. The object-type breakdown of 099.go for both CH and B1 is locals: 0.1/0.0 (reads/writes); nonvisible locals: 9.7/8.2; globals: 7.1/5.4; formals: 0.1/0.1; and heap: 0/0.

When the CH and AN/B1/B2 analyses differ it is always because of a heap-directed pointer; on all benchmarks the CH and AN/B1/B2 analyses report the same average for nonheap-directed pointers. The ST and B1 analyses can differ in all categories.

Considering the charts in Fig. 7, it seems that AN/B1/B2 are as precise as CH for pointers directed to locals, parameters, and globals. Therefore, CH, if employed at all, should focus on pointers directed to the heap.

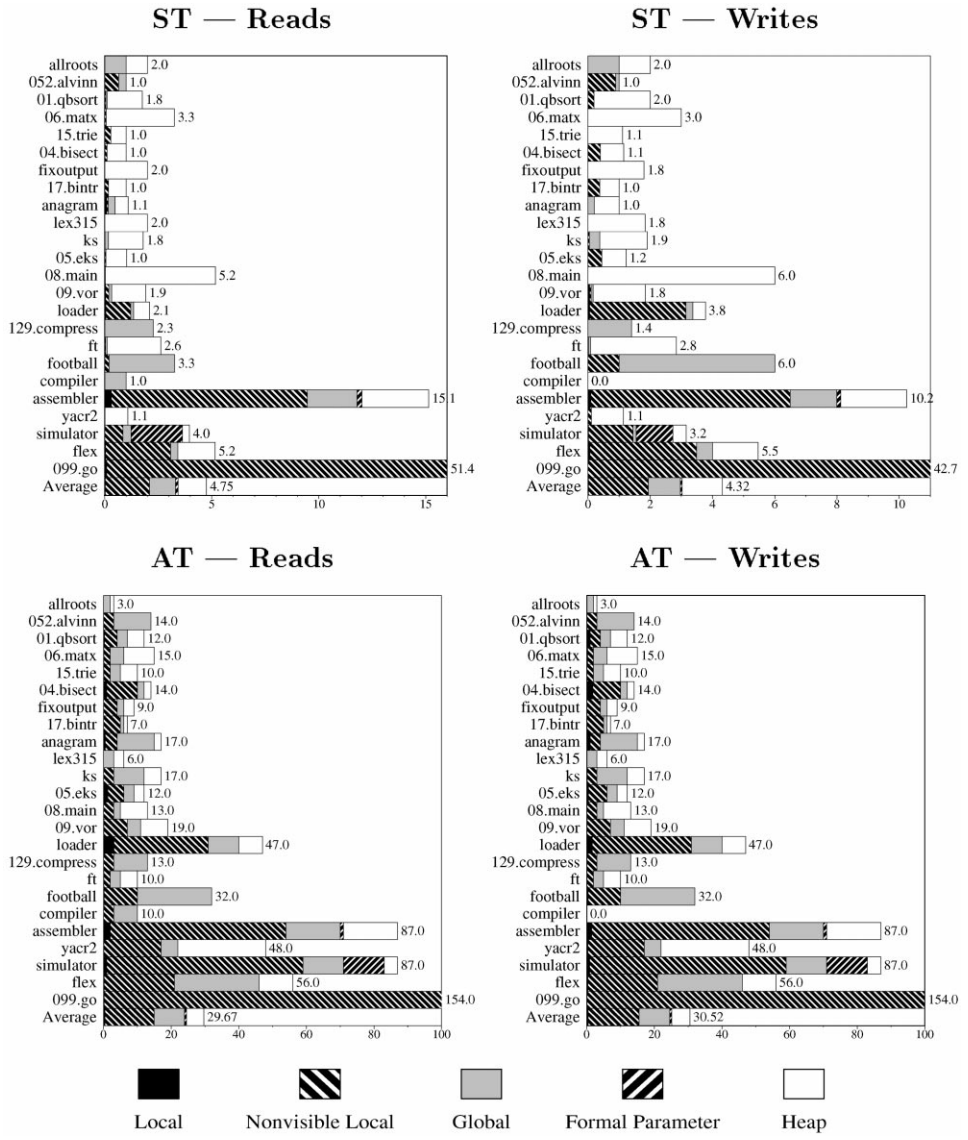


Fig. 8. Breakdown of average object type pointed to by a dereferenced pointer for the ST and AT analyses. The breakdown of 099.go for ST is locals: 0.1/0.0; nonvisible locals: 33.4/28.6; globals: 17.8/14.3; formals: 0.1/0.1; and heap: 0/0. The breakdown of 099.go for AT is locals: 0.2/0.1; nonvisible locals: 106.8/106.9; globals: 46/46; formals: 1.0/1.0; and heap: 0/0.

The precision results for `099.go` merit discussion. An average of 17.03 and 13.64 objects are returned for reads and writes, respectively, with a maximum of 100. This program contains six small list-processing functions (using an array-based "cursor" implementation) that accept a pointer to the head of a list as a parameter. One of these functions, `addlist`, is called 404 times and passed the address of 100 different

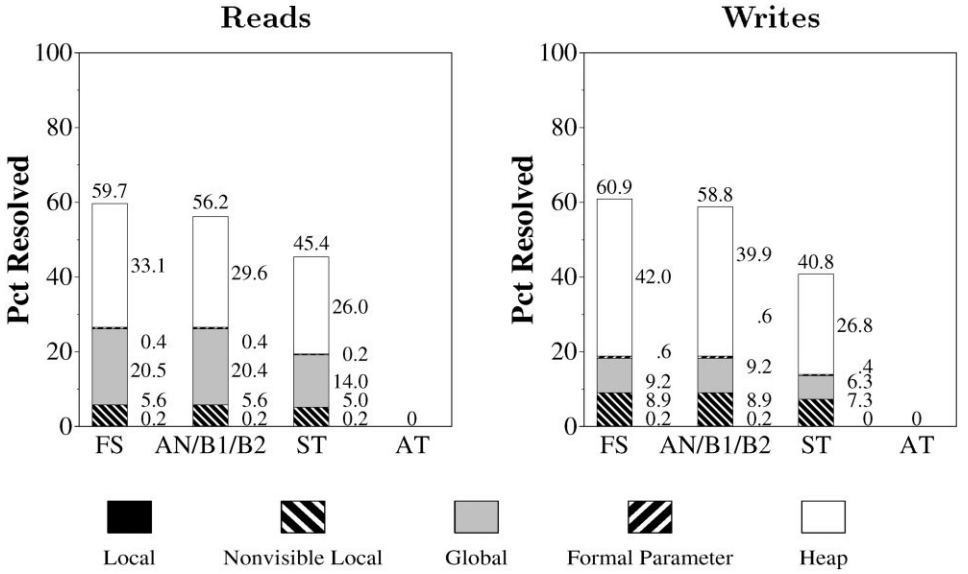


Fig. 9. Percentage of dereferenced pointers that resolve to one object in our model.

actuals for the list header, resulting in 100 aliases for the formal parameter. However, because the lifetime of the formal is limited to this function (it does not call any other function), these relations are not propagated to any other function. Therefore, these relations do not suffer the effects of the unrealizable path problem mentioned in Section 2.

Another useful precision metric is how many pointer dereferences can be resolved to exactly one object. If the object is a named variable, as opposed to a heap object, the pointer dereference could be replaced with the variable, assuming a correctly working program, i.e., a program that does not dereference the NULL pointer. The charts of Fig. 9 report the percentage of dereferenced pointers that resolve to exactly one object in our model.

As was the case with the general precision results, the charts show that the CH and AN/B1/B2 analyses have the same effectiveness for pointers directed to locals, parameters, and globals, resolving 26.6% and 18.9% of such pointers for reads and writes, respectively. Thus, they differ only in heap-directed pointers. The CH analysis resolves 3.5% and 2.1% more of the reads and writes, respectively, for heap-directed pointers. The ST analysis does compromise precision using this metric for both heap- and nonheap-directed pointers. It resolves 45.4% of the reads, compared to 59.7% (CH) and 56.2% (AN/B1/B2), and 40.8% of the writes, compared to 60.9% (CH) and 58.8% (AN/B1/B2). However, this is still a significant improvement over the next level of analysis, AT, which is unable to resolve any pointer dereferences because all benchmarks contains at least two variables or heap locations whose address are stored.

In summary, the precision of the ST analysis is considerably better than AT with little degradation of performance. The precision of the AN/B1/B2 analyses is as good as the CH analysis in many cases, with improved efficiency for B1, and sometimes for AN. Although there is a difference in precision between ST and AN/B1/B2, the significance of this difference will likely depend on how the alias information is used by subsequent analyses.

## 5. Related work

Landi et al. [31,50] report precision results for the computation of the MOD problem using the flow-sensitive context-sensitive pointer alias algorithm of Landi and Ryder [29]. Among the metrics they report is the number of "thru-deref" assigns, which corresponds to the "write" metrics reported in Fig. 6. They compare this analysis with a flow-insensitive analysis [56] that shares the property of Steensgaard's analysis [49] (called "ST" in this paper) in that it groups all objects pointed-to by a variable into an equivalence class. They found that the more precise analysis provided improved precision, but exhausted memory on some programs that the less precise analysis was able to process.

Ruf [39] presents an empirical study of two algorithms: a flow-sensitive algorithm similar to the one we have implemented, and a context-sensitive version of the same algorithm. His results showed that the context-sensitive algorithm did not improve precision for pointers where they are dereferenced, but cautioned that this may be a characteristic of the benchmark suite analyzed.

Shapiro and Horwitz [46] present an empirical comparison of four flow-insensitive algorithms. They compare implementations of AT, ST, and AN, along with an algorithm by Shapiro and Horwitz [47], which can be tuned to provide precision between ST and AN. The authors measure the precision of these analyses by implementing three data flow analyses (GMOD, live variables, and truly live variables) and an interprocedural slicing algorithm. In addition to these alias analysis clients, the authors also report the direct precision of the alias analysis algorithms in terms of the total number of points-to relations. They conclude (1) a more precise flow-insensitive analysis (AN) generally leads to increased precision by the subsequent analyses that use this information with varying magnitudes and can also improve the efficiency of subsequent analyses that use this information; (2) metrics measuring the alias analysis precision tend to be good predictors on the precision of subsequent analyses that use alias information.

Diwan et al. [12] examine the effectiveness of three type-based flow-insensitive analyses for a type-safe language (Modula-3). The first two algorithms rely on type declarations. The third considers assignments in a manner similar to the ST analysis, but retains declared type information. They evaluate the effect of these algorithms on redundant load elimination using statical, dynamic, and upper bound metrics. They conclude that for type-safe languages, such as Modula-3 or Java, a fast and simple type-based analysis may be sufficient.

Yong et al. [55] present a tunable pointer-analysis framework for handling structures in the presence of casting. They provide experimental results from four instances of the framework using a flow- and context-insensitive algorithm, which appears to be similar to the AN algorithm. Their results show that for this pointer algorithm distinguishing struct components can improve precision where pointers are dereferenced. They do not address if similar results hold for other pointer analyses.

Liang and Harrold [33] describe a context-sensitive flow-insensitive algorithm and empirically compare it to three other algorithms: ST, AN, and Landi and Ryder [29], using dereferenced writes through pointers, summary edges in a system dependence graph, and average slice size as precision metrics. They demonstrate performance and precision mostly between the AN and ST algorithms. None of the implementations handles function pointers or `setjmp`/`longjmp`.

Emami et al. [13] report precision results for a flow-sensitive context-sensitive algorithm. Wilson and Lam [53,54] present an algorithm for performing context-sensitive analysis that avoids redundant analyses of functions for similar calling contexts. Ghiya and Hendren [16] present empirical data showing how points-to and connection analyses can improve traditional transformations, array dependence testing, and program understanding. Chatterjee et al. [8] describe a technique for incorporating relevant context information into a data flow analysis and illustrate their approach for points-to analysis. Empirical results on C++ programs are provided.

Ruf [40] describes a program partitioning technique that is used for a flow-sensitive points-to analysis, achieving a storage savings of 1.3–7.2 over existing methods. Zhang et al. [57] report the effectiveness of applying different pointer aliasing algorithms to different parts of a program.

Hind and Pioli [22] expand on the work of this paper by providing precision/efficiency results for clients of pointer analysis information such as mod-ref, live variables, dead assignments, conditional constant propagation, and unreachable code. Empirical results are also presented in [36,37] for several combinations of Wegman and Zadeck's [52] conditional constant propagation algorithm and the pointer analyses we have studied, including a new algorithm that synthesizes the CH analysis and conditional constant propagation.

## 6. Conclusion

This work has described an empirical study of six pointer alias analysis algorithms that use varying degrees of flow-sensitivity. We have found that

- the AT and ST analyses are efficient in both analysis time and memory consumption, and the precision of the ST over the AT analysis, where pointers were dereferenced, makes it appealing for inclusion in production compilers;
- the AN, B1, and B2 all had the same precision;

- because of additional analysis time over B1 and lack of improved precision, the B2 analysis is not beneficial;
- although we would expect the AN analysis to be more efficient than the B1 analysis, the results of this work did not strongly back this conclusion;
- the AN and B1 analyses provided additional precision over the ST analysis in the majority of programs, particularly for the larger programs, and are identical to that of the CH analysis in 18 of 24 programs, and thus are an attractive alternative to ST when additional precision is required.

In summary, the results of this paper suggest that the ST analysis is usable in production compilers. If better precision is required, the AN or B1 analyses can be used. Further precision improvement may be obtained for the CH analysis, but it appears that other enhancements, such as a more precise modeling of aggregates or heap objects, or context-sensitivity may be required to fully realize the improved benefits of a flow-sensitive analysis.

## Acknowledgements

We thank Vivek Sarkar, Michael Burke, Michael Karasick, and Lee Nackman for their support of this work. We also thank Todd Austin, Bill Landi, and Rakesh Ghiya for making their benchmarks available. Bill Landi, Laurie Hendren, Erik Ruf, Barbara Ryder, and Bob Wilson provided useful details concerning their implementations. Michael Burke, Paul Carini, and Jong-Deok Choi provided useful discussions regarding the algorithms described in [9]. Discussions with Manuel Fähndrich led to reporting intermediate read dereferences, which were not considered in [19].

Harini Srinivasan designed and implemented the initial control flow graph builder. NPIC group members Robert Culley, Lynne Delesky, Joti Giordano, Lap Chung Lam, Giampaolo Lauria, Mark Nicosia, Joseph Perillo, Keith Sanders, Truong Vu, and Ming Wu assisted with the implementation and testing of the system. David Bacon helped with the initial design of the program call graph representation.

Michael Burke, Jong-Deok Choi, Michael Ernst, John Field, G. Ramalingam, Harini Srinivasan, Laureen Treacy, the *SAS'98* program committee, and the anonymous referees provided useful comments on earlier drafts of this paper.

## References

[1] L.O. Andersen, Program analysis and specialization for the C programming language, Ph.D. Thesis, DIKU, University of Copenhagen, May 1994. Available at ftp.diku.dk/pub/diku/semantics/papers/D-203.dvi.Z.

[2] T. Austin, Pointer-intensive benchmark suite, version 1.1. http://www.cs.wisc.edu/~austin/ptr-dist.html, 1995.

[3] S. Balan, W. Bays, Spec announces new benchmark suites cint92 and cfp92, Tech. Rep. Systems Performance Evaluation Cooperative, March 1992, SPEC Newsletter 4(1).

[4] M. Burke, P. Carini, J.-D. Choi, M. Hind, Flow-insensitive interprocedural alias analysis in the presence of pointers, in: Proc. 7th Workshop on Languages and Compilers for Parallel Computing, K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), Lecture Notes in Computer Science, Vol. 892, Springer, Berlin, 1995, pp. 234–250. Extended version published as Research Report RC 19546, IBM T.J. Watson Research Center, September 1994.

[5] M. Burke, P. Carini, J.-D. Choi, M. Hind, Interprocedural pointer alias analysis, Research Report RC 21055, IBM T.J. Watson Research Center, December 1997.

[6] P. Carini, M. Hind, H. Srinivasan, Flow-sensitive interprocedural type analysis for C++, Research Report RC 20267, IBM T.J. Watson Research Center, November 1995.

[7] D.R. Chase, M. Wegman, F. Kenneth Zadeck, Analysis of pointers and structures, SIGPLAN '90 Conf. on Programming Language Design and Implementation, June 1990, pp. 296–310; SIGPLAN Notices 25(6).

[8] R. Chatterjee, B.G. Ryder, W.A. Landi, Relevant context inference, 26th Annu. ACM SIGACT-SIGPLAN Symp. on the Principles of Programming Languages, January 1999.

[9] J.-D. Choi, M. Burke, P. Carini, Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects, 20th Annu. ACM SIGACT-SIGPLAN Symp. on the Principles of Programming Languages, January 1993, pp. 232–245.

[10] J.-D. Choi, R. Cytron, J. Ferrante, Automatic construction of sparse data flow evaluation graphs, 18th Annu. ACM Symp. on the Principles of Programming Languages, January 1991, pp. 55–66.

[11] A. Deutsch, Interprocedural may-alias analysis for pointers: beyond *k*-limiting, SIGPLAN '94 Conf. on Programming Language Design and Implementation, June 1994, pp. 230–241; SIGPLAN Notices 29(6).

[12] A. Diwan, K.S. McKinley, J.E.B. Moss, Type-based alias analysis, SIGPLAN '98 Conf. on Programming Language Design and Implementation, June 1998, pp. 106–117; SIGPLAN Notices 33(5).

[13] M. Emami, R. Ghiya, L.J. Hendren, Context-sensitive interprocedural points-to analysis in the presence of function pointers, SIGPLAN '94 Conf. on Programming Language Design and Implementation, June 1994, pp. 242–256; SIGPLAN Notices 29(6).

[14] R. Ghiya, L.J. Hendren, Connection analysis: a practical interprocedural heap analysis for C, Internat. J. Parallel Programming 24 (6) (1996) 547–578.

[15] R. Ghiya, L.J. Hendren, Is it a tree, a dag or a cyclic graph? A shape analysis for heap-directed pointers in C, 23rd Annu. ACM SIGACT-SIGPLAN Symp. on the Principles of Programming Languages, January 1996, pp. 1–15.

[16] R. Ghiya, L.J. Hendren, Putting pointer analysis to work, 25th Annu. ACM SIGACT-SIGPLAN Symp. on the Principles of Programming Languages, January 1998, pp. 121–133.

[17] L.J. Hendren, A. Nicolau, Parallelizing programs with recursive data structures, IEEE Trans. Parallel Distributed Systems 1 (1) (1990) 35–47.

[18] M. Hind, M. Burke, P. Carini, J.-D. Choi, Interprocedural pointer alias analysis, ACM Trans. Programming Languages Systems 21 (4) (1999) 848–894.

[19] M. Hind, A. Pioli, An empirical comparison of interprocedural pointer alias analyses, Research Report RC 21058, IBM T.J. Watson Research Center, December 1997. Also available as SUNY at New Paltz Technical Report #98-104.

[20] M. Hind, A. Pioli, Assessing the effects of flow-sensitivity on pointer alias analyses, in: Proc. 5th Internat. Static Analysis Symposium, G. Levi (Ed.), Lecture Notes in Computer Science, Vol. 1503, Springer, Berlin, 1998, pp. 57–81.

[21] M. Hind, A. Pioli, Traveling through Dakota: Experiences with an object-oriented program analysis system, in: TOOLS USA 2000–34th Internat. Conf. on Component and Object Technology, July 2000. Also available as Research Report 21674, IBM T.J. Watson Research Center, February 2000.

[22] M. Hind, A. Pioli, Which pointer analysis should I use?, in: ACM SIGSOFT Internat. Symp. on Software Testing and Analysis, August 2000.

[23] S. Horwitz, P. Pfeiffer, T. Reps, Dependence analysis for pointer variables, SIGPLAN '89 Conf. on Programming Language Design and Implementation, June 1989, pp. 28–40; SIGPLAN Notices 24(6).

[24] P. Hudak, A semantic model of reference counting and its abstraction, Conf. Record of the 1986 ACM Symp. of LISP and Functional Programming, August 1986, pp. 351–363.

[25] N.D. Jones, S.S. Muchnick, Flow analysis and optimization of LISP-like structures, in: S.S. Muchnick, N.D. Jones (Eds.), Program Flow Analysis: Theory and Applications, Prentice-Hall, Englewood Cliffs, NJ, 1981, pp. 102–131 (Chapter 4).

[26] M. Karasick, The architecture of Montana: An open and extensible programming environment with an incremental C++ compiler, in: ACM SIGSOFT Symp. on the Foundations of Software Engineering, November 1998, pp. 131–142.
[27] W. Landi, Undecidability of static analysis, ACM Lett. Programming Languages Systems 1 (4) (1992) 323–337.
[28] W. Landi, personal communication, October 1997.
[29] W. Landi, B. Ryder, A safe approximate algorithm for interprocedural pointer aliasing, SIGPLAN '92 Conf. on Programming Language Design and Implementation, June 1992, pp. 235–248; SIGPLAN Notices 27(6).
[30] W. Landi, B. Ryder, S. Zhang, Interprocedural modification side effect analysis with pointer aliasing, SIGPLAN'93 Conf. on Programming Language Design and Implementation, June 1993, pp. 56–67; SIGPLAN Notices 28(6).
[31] W.A. Landi, B.G. Ryder, P.A. Stocks, S. Zhang, R. Altucher, A schema for interprocedural modification side-effect analysis with pointer aliasing, Tech. Rep. DCS-TR-336, Department of Computer Science, Rutgers University, May 1998.
[32] J.R. Larus, P.N. Hilfinger, Detecting conflicts between structure accesses, SIGPLAN'88 Conf. on Programming Language Design and Implementation, 1988, pp. 21–34; SIGPLAN Notices 23(7).
[33] D. Liang, M.J. Harrold, Efficient points-to analysis for wholeprogram analysis, in: Proc. 7th European Software Engineering Conf. and ACM SIGSOFT Foundations of Software Engineering, O. Nierstrasz, M. Lemoine (Eds.), Lecture Notes in Computer Science, Vol. 1687, Springer, Berlin, September 1999, pp. 199–215.
[34] T. Marlowe, W. Landi, B. Ryder, J.-D. Choi, M. Burke, P. Carini, Pointer-induced aliasing: a clarification, SIGPLAN Notices 28 (9) (1993) 67–70.
[35] T.J. Marlowe, B.G. Ryder, M.G. Burke, Defining flow sensitivity in data flow problems, Tech. Rep. RC 20138, IBM T.J. Watson Research Center, July 1995.
[36] A. Pioli, Conditional pointer aliasing and constant propagation, Master's Thesis, SUNY at New Paltz, 1999. Available at http://www.mcs.newpaltz.edu/tr as Tech. Rep. # 99-102.
[37] A. Pioli, M. Hind, Combining interprocedural pointer analysis and conditional constant propagation, Research Report 21532, IBM T.J. Watson Research Center, March 1999. Also available as SUNY at New Paltz Technical Report # 99-103.
[38] G. Ramalingam, The undecidability of aliasing, ACM Trans. Programming Languages Systems 16 (5) (1994) 1467–1471.
[39] E. Ruf, Context-insensitive alias analysis reconsidered. SIGPLAN'95 Conference on Programming Language Design and Implementation, June 1995, pp. 13–22; SIGPLAN Notices, 30(6).
[40] E. Ruf, Partitioning dataflow analyses using types, 24th Annu. ACM SIGACT-SIGPLAN Symp. on the Principles of Programming Languages, January 1997, pp. 15–26.
[41] E. Ruf, personal communication, October 1997.
[42] C. Ruggieri, T.P. Murtagh, Lifetime analysis of dynamically allocated objects, 15th Annu. ACM Symp. on the Principles of Programming Languages, January 1988, pp. 285–293.
[43] Rutgers PROLANGS. http://www.prolangs.rutgers.edu/public.html, 1999.
[44] M. Sagiv, T. Reps, R. Wilhelm, Solving shape-analysis problems in languages with destructive updating, 23rd Annu. ACM SIGACT-SIGPLAN Symp. on the Principles of Programming Languages, January 1996, pp. 16–31.
[45] M. Sagiv, T. Reps, R. Wilhelm, Solving shape-analysis problems in languages with destructive updating, ACM Trans. on Programming Languages Systems 20 (1) (1998) 1–50.
[46] M. Shapiro, S. Horwitz, The effects of the precision of pointer analysis, in: Proc. 4th Internat. Static Analysis Symp., P.V. Hentenryck (Ed.), Lecture Notes in Computer Science, Vol. 1302, Springer, Berlin, 1997, pp. 16–34.
[47] M. Shapiro, S. Horwitz, Fast and accurate flow-insensitive point-to analysis, 24th Annu. ACM SIGACT-SIGPLAN Symp. on the Principles of Programming Languages, January 1997, pp. 1–14.
[48] SPEC. SPEC CPU95, Version 1.0, Standard Performance Evaluation Corporation, http://www.specbench/org, August 1995.
[49] B. Steensgaard, Points-to analysis in almost linear time, 23rd Annu. ACM SIGACT-SIGPLAN Symp. on the Principles of Programming Languages, January 1996, pp. 32–41.

[50] P.A. Stocks, B.G. Ryder, W.A. Landi, S. Zhang, Comparing flow and context sensitivity on the modifications-side-effects problem, Internat. Symp. on Software Testing and Analysis, March 1998, pp. 21–31.
[51] R. Tarjan, Data structures and network flow algorithms, Regional Conference Series in Applied Mathematics, CMBS, Vol. 44, SIAM, Philadelphia, PA, 1983.
[52] M.N. Wegman, F. Kenneth Zadeck, Constant propagation with conditional branches, ACM Trans. on Programming Languages and Systems 13 (2) (1991) 181–210.
[53] R.P. Wilson, Efficient context-sensitive pointer analysis for C programs, Ph.D. Thesis, Stanford University, December 1997.
[54] R.P. Wilson, M.S. Lam, Efficient context-sensitive pointer analysis for C programs, SIGPLAN'95 Conf. on Programming Language Design and Implementation, June 1995, pp. 1–12; SIGPLAN Notices 30(6).
[55] S.H. Yong, S. Horwitz, T. Reps, Pointer analysis for programs with structures and casting, SIGPLAN'99 Conf. on Programming Language Design and Implementation, 1999, pp. 91–103.
[56] S. Zhang, B.G. Ryder, W. Landi, Program decomposition for pointer aliasing: A step toward practical analyses, 4th Symp. on the Foundations of Software Engineering, October 1996, pp. 81–92.
[57] S. Zhang, B.G. Ryder, W. Landi, Experiments with combined analysis for pointer aliasing, ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, June 1998, pp. 11–18.